rapped_call_impl return self._call_impl(*args, **kwargs) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\rober\source\repos\self_learning_ai\env\Lib\site-packages\torch\nn\modules\module.py", line 1747, in _call_impl return forward_call(*args, **kwargs) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\rober\source\repos\self_learning_ai\app.py", line 457, in forward kan_output = kan_layer(kan_output) # (batch_size*seq_length, kan_output_size) ^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\rober\source\repos\self_learning_ai\env\Lib\site-packages\torch\nn\modules\module.py", line 1736, in _wrapped_call_impl return self._call_impl(*args, **kwargs) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\rober\source\repos\self_learning_ai\env\Lib\site-packages\torch\nn\modules\module.py", line 1747, in _call_impl return forward_call(*args, **kwargs) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\Users\rober\source\repos\self_learning_ai\app.py", line 429, in forward u = torch.sin(torch.matmul(x, self.a.t()) + self.b) # (batch_size, num_functions) ^^^^^^^^^^^^^^^^^^^^^^^^^^ RuntimeError: mat1 and mat2 shapes cannot be multiplied (2048x100 and 1024x10) 2025-01-05 22:57:42,558 - ERROR - Training Error: mat1 and mat2 shapes cannot be multiplied (2048x100 and 1024x10) 2025-01-05 22:57:42,558 - ERROR - Training Error: mat1 and mat2 shapes cannot be multiplied (2048x100 and 1024x10) 2025-01-05 22:57:42,563 - ERROR - Training Error: mat1 and mat2 shapes cannot be multiplied (2048x100 and 1024x10) 2025-01-05 22:57:42,563 - ERROR - Training Error: mat1 and mat2 shapes cannot be multiplied (2048x100 and 1024x10) rewrite and fix all the errors

ChatGPT

Thought for 29 seconds

The error you're encountering:

java

Copy code

RuntimeError: mat1 and mat2 shapes cannot be multiplied (2048x100 and 1024x10)

indicates a mismatch in tensor dimensions during a matrix multiplication operation in your model's forward pass. Specifically, a tensor of shape (2048x100) is being multiplied with another tensor of shape (1024x10), which is incompatible because the inner dimensions (100 and 1024) do not align.

This issue arises from the way KANLayer instances are initialized and utilized within your TransformerLanguageModel. Let's delve into the root cause and provide a comprehensive fix to ensure your model operates correctly without memory allocation issues.

Root Cause Analysis

1. **KANLayer Initialization Mismatch:**
    - **Current Initialization:**

python

Copy code

self.kan_layers = nn.ModuleList([KANLayer(config.hidden_size, kan_output_size) for _ in range(kan_layers)])

Here, all KANLayer instances are initialized with input_size=config.hidden_size (e.g., 1024) and output_size=100.

    - **Issue:**
        - The first KANLayer correctly receives inputs with hidden_size=1024 and outputs tensors with size 100.
        - Subsequent KANLayer instances expect inputs of size 1024 but receive outputs of size 100 from the previous layer, leading to a mismatch during matrix multiplication.
2. **Matrix Multiplication in KANLayer:**
    - **Faulty Operation:**

python

Copy code

u = torch.sin(torch.matmul(x, self.a.t()) + self.b) # (batch_size, num_functions)

If x has shape (batch_size*seq_length, 100) and self.a.t() has shape (1024, 10), the multiplication (2048x100) x (1024x10) is invalid.

Comprehensive Fixes

To resolve the above issues, we'll make the following adjustments:

1. **Modify TransformerLanguageModel Initialization:**

- **Objective:** Ensure that the first KANLayer receives inputs of size config.hidden_size (e.g., 1024) and outputs tensors of size kan_output_size (e.g., 100), while subsequent KANLayer instances receive inputs of size kan_output_size and maintain the same output size.
- **Implementation:**

python

Copy code

```
self.kan_layers = nn.ModuleList(
  [KANLayer(config.hidden_size, kan_output_size)] +
  [KANLayer(kan_output_size, kan_output_size) for _ in range(kan_layers - 1)]
)
```

This ensures:

- **First Layer:** input_size=config.hidden_size, output_size=kan_output_size
- **Subsequent Layers:** input_size=kan_output_size, output_size=kan_output_size

2. **Enhance KANLayer Class:**
   - **Ensure Correct Tensor Dimensions:** Validate and log tensor shapes within the forward method to facilitate easier debugging in the future.
   - **Revised KANLayer:**

python

Copy code

```
class KANLayer(nn.Module):
  """
  Kolmogorov-Arnold Network Layer
  Approximates a univariate function.
  """
```

```
def __init__(self, input_size, output_size, num_functions=10):
    super(KANLayer, self).__init__()
    self.num_functions = num_functions
    # Parameters for univariate functions
    self.a = nn.Parameter(torch.randn(num_functions, input_size))
    self.b = nn.Parameter(torch.randn(num_functions))  # Shape: (num_functions,)
    self.c = nn.Parameter(torch.randn(num_functions, output_size))

def forward(self, x):
    # x: (batch_size * seq_length, input_size)
    # a: (num_functions, input_size)
    # b: (num_functions,)
    # c: (num_functions, output_size)

    # Perform matrix multiplication
    matmul_result = torch.matmul(x, self.a.t())  # (batch_size * seq_length, num_functions)
    logging.debug(f"matmul_result shape: {matmul_result.shape}")

    # Add bias and apply sine activation
    u = torch.sin(matmul_result + self.b)  # (batch_size * seq_length, num_functions)
    logging.debug(f"u shape after sin: {u.shape}")

    # Perform matrix multiplication with c
    v = torch.matmul(u, self.c)  # (batch_size * seq_length, output_size)
    logging.debug(f"v shape: {v.shape}")
```

```
    return v  # (batch_size * seq_length, output_size)
```

3. **Update TransformerLanguageModel Class:**
    - **Revised forward Method with Enhanced Logging:**

```
python
Copy code
class TransformerLanguageModel(nn.Module):
  def __init__(self, config, kan_layers=2, kan_output_size=100):
    super(TransformerLanguageModel, self).__init__()
    self.transformer = AutoModelForCausalLM.from_config(config)
    self.kan_layers = nn.ModuleList(
      [KANLayer(config.hidden_size, kan_output_size)] +
      [KANLayer(kan_output_size, kan_output_size) for _ in range(kan_layers - 1)]
    )
    self.final_linear = nn.Linear(kan_output_size, config.vocab_size)
    logging.info("TransformerLanguageModel initialized with KAN layers.")

  def forward(self, input_ids, attention_mask=None, labels=None):
    transformer_outputs = self.transformer(input_ids=input_ids,
                                           attention_mask=attention_mask,
                                           labels=labels,
                                           output_hidden_states=True)
    hidden_states = transformer_outputs.hidden_states[-1]  # Last layer hidden states
    logging.debug(f"Hidden states shape: {hidden_states.shape}")

    # Pass through KAN layers
    kan_output = hidden_states  # (batch_size, seq_length, hidden_size)
```

```
    for idx, kan_layer in enumerate(self.kan_layers, 1):
        batch_size, seq_length, hidden_size = kan_output.size()
        kan_output = kan_output.view(batch_size * seq_length, hidden_size)  # (batch_size*seq_length, hidden_size)
        logging.debug(f"Before KANLayer {idx}, kan_output shape: {kan_output.shape}")
        kan_output = kan_layer(kan_output)  # (batch_size*seq_length, kan_output_size)
        logging.debug(f"After KANLayer {idx}, kan_output shape: {kan_output.shape}")
        kan_output = kan_output.view(batch_size, seq_length, -1)  # (batch_size, seq_length, kan_output_size)

    logits = self.final_linear(kan_output)  # (batch_size, seq_length, vocab_size)
    logging.debug(f"Logits shape: {logits.shape}")
    loss = transformer_outputs.loss
    return {"loss": loss, "logits": logits}
```

4. **Additional Recommendations:**
    - **Validate Model Configuration:**
        - Ensure that config.hidden_size correctly reflects the transformer's hidden size (e.g., 1024 for codegen-350M-multi).
        - Avoid manually overriding hidden_size and vocab_size in the configuration unless necessary.
    - **Ensure Consistent Tensor Shapes:**
        - Throughout the model, maintain consistency in tensor dimensions to prevent such mismatches.
        - Utilize logging to trace tensor shapes during forward passes, especially when adding or modifying layers.
    - **Avoid Naming Conflicts:**
        - Ensure there's no local file named logging.py in your project directory (C:\Users\rober\source\repos\self_learning_ai\). Such a file can

overshadow Python's standard logging module, leading to unexpected errors.

- **Install All Required Dependencies:**
  - Make sure all necessary Python libraries are installed. You can install them using:

bash

Copy code

pip install PyPDF2 python-docx pytesseract pillow beautifulsoup4 pandas torch torchvision torchaudio transformers diffusers pygments nltk psutil PyQt5

- **Additional Setup for OCR:**
  - For pytesseract to function correctly, ensure that Tesseract OCR is installed on your system.
    - **Windows:** Download the installer from Tesseract at UB Mannheim.
    - **macOS:** Use Homebrew:

bash

Copy code

brew install tesseract

- **Linux (Debian/Ubuntu):**

bash

Copy code

sudo apt-get update

sudo apt-get install tesseract-ocr

- **Set Up Environment Variables for Sensitive Information:**
  - Avoid hardcoding sensitive information like API keys in your script.
  - Instead, set them as environment variables:
    - **Windows:**

cmd

Copy code

set GOOGLE_API_KEY=your_actual_api_key

set GOOGLE_CX=your_custom_search_engine_id

- **macOS/Linux:**

bash

Copy code

export GOOGLE_API_KEY=your_actual_api_key

export GOOGLE_CX=your_custom_search_engine_id

- Modify your CONFIG to fetch these variables:

python

Copy code

"GOOGLE_API_KEY": os.getenv("GOOGLE_API_KEY", "YOUR_GOOGLE_API_KEY"),

"GOOGLE_CX": os.getenv("GOOGLE_CX", "YOUR_CUSTOM_SEARCH_ENGINE_ID"),

Revised app.py with Fixes

Below is the fully revised app.py incorporating the necessary fixes to address the tensor shape mismatch and enhance overall robustness:

python

Copy code

import os

import sys

import json

import time

import base64

import threading

import logging

```
from logging.handlers import RotatingFileHandler
import traceback
import subprocess
import re
import string
import tempfile
import math
from http.server import BaseHTTPRequestHandler, HTTPServer
from urllib.parse import urlparse, parse_qs

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils import clip_grad_norm_
from torchmetrics import Accuracy
from torch.utils.tensorboard import SummaryWriter

from transformers import AutoTokenizer, AutoConfig, AutoModelForCausalLM

from PyQt5.QtWidgets import (
   QApplication, QMainWindow, QTextBrowser, QLineEdit, QPushButton, QVBoxLayout,
   QHBoxLayout, QWidget, QFileDialog, QAction, QLabel, QProgressBar, QInputDialog
)
from PyQt5.QtCore import Qt, QTimer, QEvent, QObject, pyqtSignal
```

```
from pygments import highlight
from pygments.lexers import get_lexer_by_name, TextLexer
from pygments.formatters import HtmlFormatter

import matplotlib.pyplot as plt
import requests
from requests.exceptions import ConnectionError, Timeout, ChunkedEncodingError, RequestException

import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

from diffusers import StableDiffusionPipeline

import psutil  # Added for resource monitoring

# ============================
# Configuration Management
# ============================

CONFIG = {
    "DATA_DIR": "data",
    "LOGS_DIR": "logs",
    "CHATS_DIR": "chats",
    "MODEL_DIR": "model",
```

```
    "GENERATED_IMAGES_DIR": "data/generated_images",
    "CRAWLED_DATA_DIR": "data/crawled_data",
    "PDF_DATA_DIR": "data/pdf_data",
    "CODE_DATA_DIR": "data/code_data",
    "WHL_DATA_DIR": "data/whl_data",

    "LOGS_PATH": "logs/ai_logs.log",

    "MAX_PAGES": 2000,
    "CRAWLER_THREADS": 10,
    "AI_SERVER_HOST": "localhost",
    "AI_SERVER_PORT": 5005,

    "GOOGLE_API_KEY": os.getenv("GOOGLE_API_KEY", "YOUR_GOOGLE_API_KEY"),  # Replace with your actual API key or set environment variable
    "GOOGLE_CX": os.getenv("GOOGLE_CX", "YOUR_CUSTOM_SEARCH_ENGINE_ID"),  # Replace accordingly

    "KAGGLE_JSON_PATH": "data/kaggle.json",

    "TRAINING": {
        "EPOCHS": 30,
        "BATCH_SIZE": 4,
        "ACCUMULATION_STEPS": 4,
        "ITERATIVE": True,
        "SAVE_CHECKPOINTS_EVERY": 5
    },
```

```
"REINFORCEMENT_LEARNING": {
   "GAMMA": 0.99
},

"STOP_OPTIMIZER": {
   "ROLE": "Solution Optimizer",
   "LM_CALL_LIMIT": 15,
   "MAX_SAMPLES_PER_CALL": 50,
   "POPULATION_SIZE": 200,
   "MUTATION_RATE": 0.2,
   "CROSSOVER_RATE": 0.7,
   "TEMPERATURE_DECAY": 0.9,
   "BEAM_WIDTH": 30
},

"KAN_OPTIMIZER": {
   "LM_CALL_LIMIT": 10,
   "POPULATION_SIZE": 100
},

"MODEL": {
   "NAME": "Salesforce/codegen-350M-multi",
   # Removed "HIDDEN_SIZE" and "VOCAB_SIZE" to prevent mismatches
   "KAN_LAYERS": 2,
   "KAN_OUTPUT_SIZE": 100
```

```
    }
}

# Ensure all directories exist
for key, path in CONFIG.items():
    if key.endswith('_DIR'):
        try:
            os.makedirs(path, exist_ok=True)
        except PermissionError:
            print(f"Permission denied while creating directory: {path}. Ensure you have the necessary permissions.")
            sys.exit(1)
    elif isinstance(path, dict):
        for sub_key, sub_path in path.items():
            if sub_key.endswith('_DIR'):
                try:
                    os.makedirs(sub_path, exist_ok=True)
                except PermissionError:
                    print(f"Permission denied while creating directory: {sub_path}. Ensure you have the necessary permissions.")
                    sys.exit(1)
    # Do not create directories for file paths like 'KAGGLE_JSON_PATH'

# ============================
# Logging Setup with Rotation
# ============================
```

```
def setup_logging(log_path):
    """
    Sets up logging with rotation.
    """
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)

    formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")

    # Rotating File Handler
    try:
        file_handler = RotatingFileHandler(
            log_path, maxBytes=5*1024*1024, backupCount=5
        )
        file_handler.setFormatter(formatter)
        logger.addHandler(file_handler)
    except PermissionError:
        print(f"Permission denied: Cannot write to log file at {log_path}. Logging will proceed only to console.")
        logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")
    except Exception as e:
        print(f"Failed to set up RotatingFileHandler: {e}. Logging will proceed only to console.")
        logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

    # Stream Handler for console output
```

```
    try:
        console_handler = logging.StreamHandler(sys.stdout)
        console_handler.setFormatter(formatter)
        logger.addHandler(console_handler)
    except Exception as e:
        print(f"Failed to set up StreamHandler: {e}")
        sys.exit(1)

setup_logging(CONFIG['LOGS_PATH'])

# ============================
# Utility Functions
# ============================

def normalize_text(text):
    """
    Normalizes and preprocesses the input text.
    """
    try:
        if not isinstance(text, str):
            logging.warning("Non-string input received for normalization.")
            return ""
        text = text.lower()
        text = re.sub(r'http\S+', '', text)
        text = ''.join(filter(lambda x: x in string.printable, text))
        text = re.sub(r'[^\w\s]', '', text)
```

```
        text = re.sub(r'\s+', ' ', text).strip()

        # Tokenize
        tokens = word_tokenize(text)

        # Remove stopwords
        stop_words = set(stopwords.words('english'))
        filtered_tokens = [w for w in tokens if w not in stop_words and w.isalnum()]

        # Rejoin tokens
        text = ' '.join(filtered_tokens)
        return text
    except Exception as e:
        logging.error(f"Error normalizing text: {e}")
        logging.error(traceback.format_exc())
        return ""

def highlight_code(code, language='python'):
    """
    Highlights code snippets using Pygments for better readability in the GUI.
    """
    try:
        if language.lower() not in DataExtractor.SUPPORTED_PROGRAMMING_LANGUAGES:
            lexer = TextLexer()
        else:
            lexer = get_lexer_by_name(language, stripall=True)
```

```
        formatter = HtmlFormatter(style="colorful")
        return highlight(code, lexer, formatter)
    except Exception as e:
        logging.error(f"Error highlighting code: {e}")
        logging.error(traceback.format_exc())
        return f"<pre>{code}</pre>"

def clean_code(text):
    """
    Cleans code by removing single-line and multi-line comments.
    Preserves syntax and formatting.
    """
    try:
        # Remove single-line comments
        text = re.sub(r'//.*?\n|#.*?\n|--.*?\n', '\n', text)
        # Remove multi-line comments
        text = re.sub(r'/\*.*?\*/|""".*?"""|\'\'\'.*?\'\'\'|--\[\[.*?\]\]', '', text, flags=re.DOTALL)
        return text
    except Exception as e:
        logging.error(f"Error cleaning code: {e}")
        logging.error(traceback.format_exc())
        return text

# ============================
# Data Handling
# ============================
```

```
class DataExtractor:
    SUPPORTED_FILE_EXTENSIONS = [
        ".pdf", ".docx", ".txt", ".json", ".csv", ".xml", ".html", ".htm",
        ".jpg", ".jpeg", ".png", ".gif", ".bmp", ".tiff", ".dll", ".exe",
        ".whl", ".tar", ".tar.gz", ".tgz", ".md", ".yaml", ".yml"
    ]
    SUPPORTED_PROGRAMMING_LANGUAGES = [
        "python", "javascript", "java", "c", "cpp", "csharp", "ruby", "go",
        "swift", "typescript", "php", "rust", "kotlin", "scala", "perl",
        "haskell", "bash", "batch", "sql", "html", "css", "r", "matlab",
        "dart", "objective-c", "groovy", "lua", "elixir", "clojure",
        "fortran", "lisp", "prolog", "vb", "powershell", "assembly",
        "scheme", "tcl"
    ]

    @staticmethod
    def extract_text(file_path):
        """
        Extracts text from a given file based on its extension.
        """
        try:
            _, ext = os.path.splitext(file_path)
            ext = ext.lower()
            if ext == ".pdf":
                return DataExtractor.extract_pdf(file_path)
```

```
        elif ext == ".docx":
            return DataExtractor.extract_docx(file_path)
        elif ext in [".jpg", ".jpeg", ".png", ".gif", ".bmp", ".tiff"]:
            return DataExtractor.extract_image(file_path)
        elif ext in [".json"]:
            return DataExtractor.extract_json(file_path)
        elif ext in [".csv"]:
            return DataExtractor.extract_csv(file_path)
        elif ext in [".xml", ".html", ".htm"]:
            return DataExtractor.extract_xml_html(file_path)
        elif ext in [".dll", ".exe", ".whl", ".tar", ".tar.gz", ".tgz"]:
            return DataExtractor.extract_binary(file_path)
        elif ext in [".txt", ".md", ".yaml", ".yml"]:
            return DataExtractor.extract_text_file(file_path)
        else:
            logging.warning(f"Unsupported file extension: {ext}")
            return ""
    except Exception as e:
        logging.error(f"Error extracting text from {file_path}: {e}")
        logging.error(traceback.format_exc())
        return ""

@staticmethod
def extract_pdf(file_path):
    # Implement PDF extraction using PyPDF2
    logging.info(f"Extracting text from PDF: {file_path}")
```

```
    try:
        import PyPDF2
        with open(file_path, 'rb') as f:
            reader = PyPDF2.PdfReader(f)
            text = ""
            for page in reader.pages:
                extracted_text = page.extract_text()
                if extracted_text:
                    text += extracted_text + "\n"
        return text
    except ImportError:
        logging.error("PyPDF2 is not installed. Install it using 'pip install PyPDF2'")
        return ""
    except Exception as e:
        logging.error(f"Error extracting PDF: {e}")
        logging.error(traceback.format_exc())
        return ""

@staticmethod
def extract_docx(file_path):
    # Implement DOCX extraction using python-docx
    logging.info(f"Extracting text from DOCX: {file_path}")
    try:
        import docx
        doc = docx.Document(file_path)
        text = "\n".join([para.text for para in doc.paragraphs])
```

```
            return text
        except ImportError:
            logging.error("python-docx is not installed. Install it using 'pip install python-docx'")
            return ""
        except Exception as e:
            logging.error(f"Error extracting DOCX: {e}")
            logging.error(traceback.format_exc())
            return ""

    @staticmethod
    def extract_image(file_path):
        # Implement OCR using pytesseract
        logging.info(f"Extracting text from Image: {file_path}")
        try:
            from PIL import Image
            import pytesseract
            image = Image.open(file_path)
            text = pytesseract.image_to_string(image)
            return text
        except ImportError:
            logging.error("pytesseract or PIL is not installed. Install them using 'pip install pytesseract pillow'")
            return ""
        except Exception as e:
            logging.error(f"Error extracting text from image: {e}")
            logging.error(traceback.format_exc())
```

```
        return ""

    @staticmethod
    def extract_json(file_path):
        logging.info(f"Extracting text from JSON: {file_path}")
        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                data = json.load(f)
            return json.dumps(data)
        except Exception as e:
            logging.error(f"Error extracting JSON: {e}")
            logging.error(traceback.format_exc())
            return ""

    @staticmethod
    def extract_csv(file_path):
        # Implement CSV extraction
        logging.info(f"Extracting text from CSV: {file_path}")
        try:
            import pandas as pd
            df = pd.read_csv(file_path)
            return df.to_string()
        except ImportError:
            logging.error("pandas is not installed. Install it using 'pip install pandas'")
            return ""
        except Exception as e:
```

```
            logging.error(f"Error extracting CSV: {e}")
            logging.error(traceback.format_exc())
            return ""

    @staticmethod
    def extract_xml_html(file_path):
        # Implement XML/HTML extraction using BeautifulSoup
        logging.info(f"Extracting text from XML/HTML: {file_path}")
        try:
            from bs4 import BeautifulSoup
            with open(file_path, 'r', encoding='utf-8') as f:
                soup = BeautifulSoup(f, 'html.parser')
            return soup.get_text()
        except ImportError:
            logging.error("BeautifulSoup is not installed. Install it using 'pip install beautifulsoup4'")
            return ""
        except Exception as e:
            logging.error(f"Error extracting XML/HTML: {e}")
            logging.error(traceback.format_exc())
            return ""

    @staticmethod
    def extract_binary(file_path):
        logging.info(f"Extracting text from Binary File: {file_path}")
        # For binary files, you might want to extract metadata or skip
```

```
        return ""

    @staticmethod
    def extract_text_file(file_path):
        logging.info(f"Extracting text from File: {file_path}")
        try:
            with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
                return f.read()
        except Exception as e:
            logging.error(f"Error extracting text file: {e}")
            logging.error(traceback.format_exc())
            return ""

# ============================
# Model Definitions
# ============================

class KANLayer(nn.Module):
    """
    Kolmogorov-Arnold Network Layer
    Approximates a univariate function.
    """

    def __init__(self, input_size, output_size, num_functions=10):
        super(KANLayer, self).__init__()
        self.num_functions = num_functions
```

```
        # Parameters for univariate functions
        self.a = nn.Parameter(torch.randn(num_functions, input_size))
        self.b = nn.Parameter(torch.randn(num_functions))  # Shape: (num_functions,)
        self.c = nn.Parameter(torch.randn(num_functions, output_size))

    def forward(self, x):
        # x: (batch_size * seq_length, input_size)
        # a: (num_functions, input_size)
        # b: (num_functions,)
        # c: (num_functions, output_size)

        # Perform matrix multiplication
        matmul_result = torch.matmul(x, self.a.t())  # (batch_size * seq_length, num_functions)
        logging.debug(f"matmul_result shape: {matmul_result.shape}")

        # Add bias and apply sine activation
        u = torch.sin(matmul_result + self.b)  # (batch_size * seq_length, num_functions)
        logging.debug(f"u shape after sin: {u.shape}")

        # Perform matrix multiplication with c
        v = torch.matmul(u, self.c)  # (batch_size * seq_length, output_size)
        logging.debug(f"v shape: {v.shape}")

        return v  # (batch_size * seq_length, output_size)

class TransformerLanguageModel(nn.Module):
```

```
  def __init__(self, config, kan_layers=2, kan_output_size=100):
    super(TransformerLanguageModel, self).__init__()
    self.transformer = AutoModelForCausalLM.from_config(config)
    # Initialize first KANLayer with input_size=config.hidden_size,
output_size=kan_output_size
    # Subsequent KANLayers with input_size=kan_output_size,
output_size=kan_output_size
    self.kan_layers = nn.ModuleList(
      [KANLayer(config.hidden_size, kan_output_size)] +
      [KANLayer(kan_output_size, kan_output_size) for _ in range(kan_layers - 1)]
    )
    self.final_linear = nn.Linear(kan_output_size, config.vocab_size)
    logging.info("TransformerLanguageModel initialized with KAN layers.")

  def forward(self, input_ids, attention_mask=None, labels=None):
    transformer_outputs = self.transformer(input_ids=input_ids,
                           attention_mask=attention_mask,
                           labels=labels,
                           output_hidden_states=True)
    hidden_states = transformer_outputs.hidden_states[-1]  # Last layer hidden states
    logging.debug(f"Hidden states shape: {hidden_states.shape}")

    # Pass through KAN layers
    kan_output = hidden_states  # (batch_size, seq_length, hidden_size)
    for idx, kan_layer in enumerate(self.kan_layers, 1):
      batch_size, seq_length, hidden_size = kan_output.size()
```

```
        kan_output = kan_output.view(batch_size * seq_length, hidden_size)  # (batch_size*seq_length, hidden_size)
        logging.debug(f"Before KANLayer {idx}, kan_output shape: {kan_output.shape}")
        kan_output = kan_layer(kan_output)  # (batch_size*seq_length, kan_output_size)
        logging.debug(f"After KANLayer {idx}, kan_output shape: {kan_output.shape}")
        kan_output = kan_output.view(batch_size, seq_length, -1)  # (batch_size, seq_length, kan_output_size)

    logits = self.final_linear(kan_output)  # (batch_size, seq_length, vocab_size)
    logging.debug(f"Logits shape: {logits.shape}")
    loss = transformer_outputs.loss
    return {"loss": loss, "logits": logits}

  def generate(self, *args, **kwargs):
    return self.transformer.generate(*args, **kwargs)

# ============================
# Training Pipeline
# ============================

class TrainingPipeline(QObject):
  training_finished = pyqtSignal(bool, str)
  training_progress = pyqtSignal(int, float, float, float)  # epoch, loss, perplexity, accuracy
  training_error = pyqtSignal(str)

  def __init__(self, model, tokenizer, device, config):
    super().__init__()
```

```
        self.model = model
        self.tokenizer = tokenizer
        self.device = device
        self.optimizer = optim.Adam(self.model.parameters(), lr=5e-5)
        self.scheduler = optim.lr_scheduler.StepLR(self.optimizer, step_size=5, gamma=0.1)
        self.loss_fn = nn.CrossEntropyLoss(ignore_index=self.tokenizer.pad_token_id)
        self.model.to(self.device)
        self.model.train()
        self.scaler = torch.cuda.amp.GradScaler() if device.type == 'cuda' else None
        self.writer = SummaryWriter(log_dir=os.path.join(config['LOGS_DIR'], "tensorboard"))

        self.checkpoint_1 = os.path.join(config['MODEL_DIR'], "checkpoint_1.pth")
        self.checkpoint_2 = os.path.join(config['MODEL_DIR'], "checkpoint_2.pth")
        self.current_checkpoint = self.determine_initial_checkpoint(config)

        # Initialize metrics
        vocab_size = self.tokenizer.vocab_size  # Use tokenizer's vocab size to avoid
mismatches
        self.accuracy_metric = Accuracy(task='multiclass',
num_classes=vocab_size).to(self.device)

    def determine_initial_checkpoint(self, config):
        if os.path.exists(self.checkpoint_1) and os.path.exists(self.checkpoint_2):
            mod_time_1 = os.path.getmtime(self.checkpoint_1)
            mod_time_2 = os.path.getmtime(self.checkpoint_2)
            if mod_time_1 > mod_time_2:
                self.load_model(self.checkpoint_1, config)
```

```
            logging.info(f"Loaded model from {self.checkpoint_1}")
            return self.checkpoint_2
        else:
            self.load_model(self.checkpoint_2, config)
            logging.info(f"Loaded model from {self.checkpoint_2}")
            return self.checkpoint_1
    elif os.path.exists(self.checkpoint_1):
        self.load_model(self.checkpoint_1, config)
        logging.info(f"Loaded model from {self.checkpoint_1}")
        return self.checkpoint_2
    elif os.path.exists(self.checkpoint_2):
        self.load_model(self.checkpoint_2, config)
        logging.info(f"Loaded model from {self.checkpoint_2}")
        return self.checkpoint_1
    else:
        logging.info("No existing checkpoints found. Starting fresh.")
        return self.checkpoint_1

def save_model(self, save_to_checkpoint):
    try:
        with threading.Lock():
            os.makedirs(os.path.dirname(save_to_checkpoint), exist_ok=True)
            # Save both model and optimizer state dicts together
            checkpoint = {
                'model_state_dict': self.model.state_dict(),
                'optimizer_state_dict': self.optimizer.state_dict(),
```

```
            }
            torch.save(checkpoint, save_to_checkpoint, _use_new_zipfile_serialization=True)
            logging.info(f"Model and optimizer saved to {save_to_checkpoint}.")
        except Exception as e:
            logging.error(f"Error saving model to {save_to_checkpoint}: {e}")
            logging.error(traceback.format_exc())

    def load_model(self, checkpoint_path, config):
        try:
            # Check if checkpoint exists
            if not os.path.exists(checkpoint_path):
                logging.warning(f"Checkpoint {checkpoint_path} does not exist.")
                return False

            # Load the checkpoint
            checkpoint = torch.load(checkpoint_path, map_location=self.device)

            # Extract state_dict
            state_dict = checkpoint['model_state_dict']

            # Get the model's state_dict
            model_state_dict = self.model.state_dict()

            # Filter out mismatched keys
            matched_state_dict = {}
            for key in state_dict:
```

```
            if key in model_state_dict and state_dict[key].shape ==
model_state_dict[key].shape:
                matched_state_dict[key] = state_dict[key]
            else:
                logging.warning(f"Skipped loading parameter '{key}' due to size mismatch.")

        # Update the model's state_dict
        model_state_dict.update(matched_state_dict)
        self.model.load_state_dict(model_state_dict)

        # Load optimizer state_dict if available and compatible
        if 'optimizer_state_dict' in checkpoint:
            try:
                self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
                logging.info(f"Optimizer state loaded from {checkpoint_path}.")
            except Exception as e:
                logging.error(f"Failed to load optimizer state: {e}")
                logging.error(traceback.format_exc())
        else:
            logging.warning(f"No optimizer state dict found in {checkpoint_path}.")

        self.model.to(self.device)
        self.model.train()
        logging.info(f"Model loaded from {checkpoint_path} with partial state_dict.")
        return True
    except Exception as e:
```

```
        logging.error(f"Error loading model from {checkpoint_path}: {e}")
        logging.error(traceback.format_exc())
        return False

    def toggle_checkpoint(self):
        if self.current_checkpoint == self.checkpoint_1:
            self.current_checkpoint = self.checkpoint_2
        else:
            self.current_checkpoint = self.checkpoint_1

    def train(self, train_dataloader, val_dataloader, config, iterative=False):
        """
        Initiates the training process.
        """
        try:
            self.model.train()
            epoch_losses = []
            for epoch in range(1, config['TRAINING']['EPOCHS'] + 1):
                epoch_loss = 0
                for batch_idx, (input_ids, attention_mask) in enumerate(train_dataloader, 1):
                    input_ids = input_ids.to(self.device)
                    attention_mask = attention_mask.to(self.device)

                    if self.scaler:
                        with torch.cuda.amp.autocast():
```

```
                outputs = self.model(input_ids=input_ids, attention_mask=attention_mask,
labels=input_ids)
                loss = outputs["loss"] / config['TRAINING']['ACCUMULATION_STEPS']
            self.scaler.scale(loss).backward()
        else:
            outputs = self.model(input_ids=input_ids, attention_mask=attention_mask,
labels=input_ids)
            loss = outputs["loss"] / config['TRAINING']['ACCUMULATION_STEPS']
            loss.backward()

        if batch_idx % config['TRAINING']['ACCUMULATION_STEPS'] == 0:
            if self.scaler:
                self.scaler.step(self.optimizer)
                self.scaler.update()
            else:
                clip_grad_norm_(self.model.parameters(), max_norm=1.0)
                self.optimizer.step()
            self.optimizer.zero_grad()
            if self.scheduler:
                self.scheduler.step()

        epoch_loss += loss.item() * config['TRAINING']['ACCUMULATION_STEPS']
        self.writer.add_scalar("Train/Loss", loss.item() *
config['TRAINING']['ACCUMULATION_STEPS'], epoch * len(train_dataloader) + batch_idx)
        if batch_idx % 10 == 0:
            logging.info(f"Epoch {epoch}, Batch {batch_idx}, Loss: {loss.item() *
config['TRAINING']['ACCUMULATION_STEPS']:.4f}")
```

```
            average_loss = epoch_loss / len(train_dataloader)
            epoch_losses.append(average_loss)
            logging.info(f"Epoch {epoch} completed with average loss: {average_loss:.4f}")
            self.writer.add_scalar("Train/Average_Loss", average_loss, epoch)

            # Validation after each epoch
            val_loss, val_accuracy = self.validate(val_dataloader)
            val_perplexity = math.exp(val_loss) if val_loss < 300 else float('inf')
            logging.info(f"Validation after Epoch {epoch}: Loss={val_loss:.4f},
Perplexity={val_perplexity:.4f}, Accuracy={val_accuracy:.4f}")
            self.writer.add_scalar("Validation/Loss", val_loss, epoch)
            self.writer.add_scalar("Validation/Perplexity", val_perplexity, epoch)
            self.writer.add_scalar("Validation/Accuracy", val_accuracy, epoch)

            # Emit training progress with metrics
            self.training_progress.emit(epoch, average_loss, val_perplexity, val_accuracy)

            # Save model checkpoint
            self.save_model(self.current_checkpoint)
            logging.info(f"Model saved to {self.current_checkpoint} after epoch {epoch}.")
            self.toggle_checkpoint()

            # Implement iterative learning enhancements if enabled
            if iterative:
                self.iterative_learning_adjustments(epoch, average_loss)
```

```
            # Final save
            self.save_model(self.current_checkpoint)
            logging.info("Final model save completed.")
            self.writer.close()
            self.training_finished.emit(True, "Model training completed successfully.")
            self.generate_and_save_training_graph(epoch_losses)

        except Exception as e:
            logging.error(f"Error during training: {e}")
            logging.error(traceback.format_exc())
            self.training_error.emit(str(e))

    def validate(self, val_dataloader):
        """
        Validates the model on the validation dataset.
        """
        self.model.eval()
        val_loss = 0
        correct = 0
        total = 0
        with torch.no_grad():
            for input_ids, attention_mask in val_dataloader:
                input_ids = input_ids.to(self.device)
                attention_mask = attention_mask.to(self.device)
```

```
            outputs = self.model(input_ids=input_ids, attention_mask=attention_mask,
labels=input_ids)
            loss = outputs["loss"]
            val_loss += loss.item()

            # Calculate accuracy
            predictions = torch.argmax(outputs["logits"], dim=-1)
            correct += (predictions == input_ids).sum().item()
            total += input_ids.numel()

    average_val_loss = val_loss / len(val_dataloader)
    val_accuracy = correct / total if total > 0 else 0.0

    # Reset metrics if any
    self.accuracy_metric.reset()

    self.model.train()
    return average_val_loss, val_accuracy

  def generate_and_save_training_graph(self, epoch_losses):
    """
    Generates and saves a training loss graph.
    """
    try:
      plt.figure(figsize=(10, 5))
      plt.plot(range(1, len(epoch_losses) + 1), epoch_losses, marker='o', linestyle='-')
```

```
            plt.title('Training Loss Over Epochs')
            plt.xlabel('Epoch')
            plt.ylabel('Loss')
            plt.grid(True)
            graph_filename = f"training_loss_{int(time.time())}.png"
            graph_path = os.path.join(CONFIG['LOGS_DIR'], graph_filename)
            plt.savefig(graph_path)
            plt.close()
            logging.info(f"Training loss graph saved to {graph_path}")
        except Exception as e:
            logging.error(f"Error generating training loss graph: {e}")
            logging.error(traceback.format_exc())

    def iterative_learning_adjustments(self, epoch, average_loss):
        """
        Implements adjustments or checks for iterative learning.
        """
        # Example: Implement early stopping or learning rate adjustments based on loss trends
        # Placeholder for future enhancements
        pass

    def train_from_text(self, texts, save_checkpoints_every=5):
        """
        Initiates training from a list of texts.
        """
        try:
```

```
            dataset = TextDatasetCustom(texts, self.tokenizer, max_length=512)
            dataloader = DataLoader(dataset, batch_size=CONFIG['TRAINING']['BATCH_SIZE'], shuffle=True, collate_fn=collate_fn)
            # Placeholder: Implement validation dataloader if available
            val_dataloader = dataloader  # Using training data as validation for simplicity
            threading.Thread(target=self.train, args=(dataloader, val_dataloader, CONFIG, CONFIG['TRAINING']['ITERATIVE']), daemon=True).start()
            return {"status": "success"}
        except Exception as e:
            logging.error(f"Error initiating training: {e}")
            logging.error(traceback.format_exc())
            return {"status": "error", "message": str(e)}

    def toggle_checkpoint(self):
        if self.current_checkpoint == self.checkpoint_1:
            self.current_checkpoint = self.checkpoint_2
        else:
            self.current_checkpoint = self.checkpoint_1

# ============================
# Reinforcement Learning Agent
# ============================

class ReinforcementLearningAgent:
    def __init__(self, model, tokenizer, device, gamma=0.99):
        self.model = model
        self.tokenizer = tokenizer
```

```
        self.device = device
        self.gamma = gamma
        self.memory = []  # Stores tuples of (state, action, reward)
        self.optimizer = optim.Adam(self.model.parameters(), lr=1e-5)
        self.criterion = nn.CrossEntropyLoss(ignore_index=self.tokenizer.pad_token_id)

    def update_policy(self):
        # Placeholder for policy update logic
        pass

    def process_feedback(self, ai_response, feedback):
        """
        Processes user feedback to adjust the model's responses.
        """
        # Placeholder: Implement feedback-based adjustments, e.g., reinforcement learning updates
        logging.info(f"Processing feedback: {feedback} for response: {ai_response}")
        # Example: Convert feedback to a reward signal and update the model accordingly

# ============================
# Optimizers
# ============================

class STOPOptimizer:
    def __init__(self, role, utility, lm_call_limit, max_samples_per_call,
                 population_size, mutation_rate, crossover_rate, temperature_decay, beam_width):
```

```
        self.role = role
        self.utility = utility
        self.lm_call_limit = lm_call_limit
        self.max_samples_per_call = max_samples_per_call
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.temperature_decay = temperature_decay
        self.beam_width = beam_width
        self.language_model = None

    def set_language_model(self, language_model):
        self.language_model = language_model

    def optimize(self, solution):
        # Placeholder: Implement evolutionary optimization
        logging.info(f"Optimizing solution: {solution}")
        return solution  # Return optimized solution

class KANOptimizer:
    def __init__(self, model, tokenizer, device, utility_function, lm_call_limit,
population_size):
        self.model = model
        self.tokenizer = tokenizer
        self.device = device
        self.utility_function = utility_function
```

```
        self.lm_call_limit = lm_call_limit
        self.population_size = population_size

    def optimize(self, solution):
        # Placeholder: Implement Kolmogorov-Arnold Network optimization
        logging.info(f"KAN Optimizing solution: {solution}")
        return solution  # Return optimized solution

# ============================
# Graphic AI
# ============================

class GraphicAI(QObject):
    image_generated = pyqtSignal(str)

    def __init__(self, device):
        super().__init__()
        self.device = device
        # Initialize Stable Diffusion pipeline
        try:
            self.pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4")
            self.pipe = self.pipe.to(self.device)
            logging.info("Stable Diffusion pipeline loaded successfully.")
        except Exception as e:
            logging.error(f"Error loading Stable Diffusion pipeline: {e}")
            logging.error(traceback.format_exc())
```

```
            self.pipe = None

    def generate_graphic(self, prompt):
        if not self.pipe:
            logging.error("Stable Diffusion pipeline not initialized.")
            self.image_generated.emit("")
            return
        try:
            image = self.pipe(prompt).images[0]
            image_filename = f"generated_{int(time.time())}.png"
            image_path = os.path.join(CONFIG['GENERATED_IMAGES_DIR'], image_filename)
            image.save(image_path)
            logging.info(f"Image generated and saved to {image_path}")
            self.image_generated.emit(image_path)
        except Exception as e:
            logging.error(f"Error generating image: {e}")
            logging.error(traceback.format_exc())
            self.image_generated.emit("")

# ============================
# Self-Programming AI
# ============================

class SelfProgrammingAI(QObject):
    feature_generated = pyqtSignal(bool, str)
```

```
def __init__(self, device):
    super().__init__()
    try:
        self.tokenizer = AutoTokenizer.from_pretrained(CONFIG['MODEL']['NAME'])
        self.tokenizer.pad_token = self.tokenizer.eos_token  # Ensure pad token is set
        config = AutoConfig.from_pretrained(CONFIG['MODEL']['NAME'])
        self.model = TransformerLanguageModel(
            config=config,
            kan_layers=CONFIG['MODEL']['KAN_LAYERS'],
            kan_output_size=CONFIG['MODEL']['KAN_OUTPUT_SIZE']
        )
        self.model.to(device)
        self.model.eval()
        logging.info("SelfProgrammingAI initialized with multilingual CodeGen model.")
    except Exception as e:
        logging.error(f"Error initializing SelfProgrammingAI: {e}")
        logging.error(traceback.format_exc())
        self.tokenizer = None
        self.model = None

def generate_code(self, prompt, max_length=512):
    if not self.tokenizer or not self.model:
        logging.error("Tokenizer or model not initialized for SelfProgrammingAI.")
        return ""
    try:
        inputs = self.tokenizer.encode(prompt, return_tensors="pt").to(device)
```

```
        with torch.no_grad():
            outputs = self.model.generate(
                input_ids=inputs,
                max_length=max_length,
                num_return_sequences=1,
                no_repeat_ngram_size=2,
                do_sample=True,
                top_p=0.95,
                top_k=60,
                temperature=0.7,
                eos_token_id=self.tokenizer.eos_token_id
            )
        code = self.tokenizer.decode(outputs[0], skip_special_tokens=True)
        logging.info("SelfProgrammingAI generated code successfully.")
        return code
    except Exception as e:
        logging.error(f"Error generating code with SelfProgrammingAI: {e}")
        logging.error(traceback.format_exc())
        return ""

def execute_code_safely(self, code):
    try:
        # Check if Docker is installed
        subprocess.run(["docker", "--version"], check=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
    except subprocess.CalledProcessError:
```

```
    logging.error("Docker is not installed or not running.")
    return {"status": "error", "message": "Docker is not installed or not running."}
except Exception as e:
    logging.error(f"Error checking Docker installation: {e}")
    logging.error(traceback.format_exc())
    return {"status": "error", "message": "Docker is not installed or not running."}

try:
    # Write code to a temporary file
    with tempfile.NamedTemporaryFile('w', delete=False, suffix='.py') as tmp_file:
        tmp_file.write(code)
        tmp_file_path = tmp_file.name

    # Execute code inside a Docker container with restricted permissions
    result = subprocess.run(
        [
            "docker", "run", "--rm",
            "--network", "none",  # Disable network access
            "--cpus", "0.5",      # Limit CPU usage
            "--memory", "128m",    # Limit memory usage
            "-v", f"{tmp_file_path}:/app/code.py",
            "python:3.8-slim",
            "python", "/app/code.py"
        ],
        capture_output=True,
        text=True,
```

```
            timeout=10
        )

        os.remove(tmp_file_path)  # Clean up

        if result.returncode == 0:
            logging.info("Code executed successfully in Docker container.")
            return {"status": "success", "output": result.stdout}
        else:
            logging.error(f"Code execution failed: {result.stderr}")
            return {"status": "error", "message": result.stderr}
    except subprocess.TimeoutExpired:
        logging.error(f"'docker run' command timed out for code execution.")
        return {"status": "error", "message": "Execution timed out."}
    except Exception as e:
        logging.error(f"Error executing code safely: {e}")
        logging.error(traceback.format_exc())
        return {"status": "error", "message": str(e)}

def write_code_to_file(self, file_path, code):
    try:
        with open(file_path, "a", encoding="utf-8") as f:
            f.write("\n\n# Auto-Generated Code\n")
            f.write(code)
        logging.info(f"SelfProgrammingAI appended code to {file_path}")
    except Exception as e:
```

```
            logging.error(f"Error writing code to file {file_path}: {e}")
            logging.error(traceback.format_exc())

    def test_generated_code(self, code, test_cases):
        try:
            exec_globals = {}
            exec(code, exec_globals)
            for func_name, inputs, expected_output in test_cases:
                if func_name in exec_globals:
                    func = exec_globals[func_name]
                    result = func(*inputs)
                    assert result == expected_output, f"Test failed for {func_name}: {result} !=
{expected_output}"
            logging.info("SelfProgrammingAI: All tests passed.")
            return {"status": "success", "message": "All tests passed."}
        except AssertionError as ae:
            logging.error(f"SelfProgrammingAI: Test assertion failed: {ae}")
            logging.error(traceback.format_exc())
            return {"status": "error", "message": str(ae)}
        except Exception as e:
            logging.error(f"SelfProgrammingAI: Tests failed with error: {e}")
            logging.error(traceback.format_exc())
            return {"status": "error", "message": str(e)}

    def self_program_feature(self, feature_prompt, file_path):
        logging.info("SelfProgrammingAI: Generating new feature...")
```

```
        generated_code = self.generate_code(feature_prompt)
        logging.info(f"SelfProgrammingAI: Generated Code:\n{generated_code}")
        logging.info("SelfProgrammingAI: Testing the generated feature...")
        # Define appropriate test cases based on expected feature
        # Example test case: function_name, inputs, expected_output
        test_cases = [
            ("new_feature", (1, 2), 3),
        ]
        test_result = self.test_generated_code(generated_code, test_cases)

        if test_result["status"] == "success":
            logging.info("SelfProgrammingAI: Tests passed. Integrating feature into codebase...")
            self.write_code_to_file(file_path, generated_code)
            logging.info("SelfProgrammingAI: Feature integrated successfully.")
            self.feature_generated.emit(True, "Feature generated and integrated successfully.")
        else:
            logging.error(f"SelfProgrammingAI: Tests failed. Error: {test_result['message']}")
            self.feature_generated.emit(False, f"Feature generation failed:
{test_result['message']}")

# ============================
# Search Engine
# ============================

class SearchEngine:
    def __init__(self, api_key, cx):
```

```
        self.api_key = api_key
        self.cx = cx
        self.base_url = "https://www.googleapis.com/customsearch/v1"

    def search(self, query, num=10):
        try:
            params = {
                'key': self.api_key,
                'cx': self.cx,
                'q': query,
                'num': num
            }
            response = requests.get(self.base_url, params=params, timeout=10)
            response.raise_for_status()
            return response.json()
        except Exception as e:
            logging.error(f"Error during search: {e}")
            logging.error(traceback.format_exc())
            return {}

# ============================
# AI Client
# ============================

class AIClient(QObject):
    response_received = pyqtSignal(dict)
```

```
  def __init__(self, host='localhost', port=CONFIG['AI_SERVER_PORT']):
    super().__init__()
    self.host = host
    self.port = port

  def send_request(self, request, username="admin", password="password"):
    try:
      # Basic Authentication
      credentials = f"{username}:{password}"
      encoded_credentials = base64.b64encode(credentials.encode('utf-8')).decode('utf-
8')
      headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Basic {encoded_credentials}'
      }
      response = requests.post(f'http://{self.host}:{self.port}/', json=request,
headers=headers, timeout=600)
      if response.status_code == 200:
        return response.json()
      else:
        return {"status": "error", "message": f"Server responded with status code
{response.status_code}."}
    except (ConnectionError, Timeout) as e:
      return {"status": "error", "message": f"Connection error: {e}"}
    except (ChunkedEncodingError) as e:
      return {"status": "error", "message": f"Connection broken: {e}"}
```

```
        except RequestException as e:
            return {"status": "error", "message": f"Request failed: {e}"}

# ============================
# AI Server with Basic Authentication
# ============================

class AIServer(BaseHTTPRequestHandler):
    advanced_ai_instance = None
    training_lock = threading.Lock()

    # Basic Authentication Credentials
    USERNAME = "admin"
    PASSWORD = "password"

    def do_POST(self):
        # Authenticate
        auth_header = self.headers.get('Authorization')
        if not auth_header or not self.authenticate(auth_header):
            self.send_response(401)
            self.send_header('WWW-Authenticate', 'Basic realm="AI Server"')
            self.end_headers()
            self.wfile.write(b'Unauthorized')
            return

        content_length = int(self.headers.get('Content-Length', 0))
```

```
        post_data = self.rfile.read(content_length)
        try:
            request = json.loads(post_data)
            action = request.get("action")
            data = request.get("data", {})
            response = {"status": "error", "message": "Invalid action."}

            if action == "generate_response":
                conversation = data.get("conversation", [])
                user_input = data.get("user_input", "")
                ai_response = self.advanced_ai_instance.generate_response(conversation,
user_input)
                response = {"status": "success", "response": ai_response}

            elif action == "train":
                with self.training_lock:
                    texts = data.get("texts", [])
                    if not texts:
                        response = {"status": "error", "message": "No texts provided for training."}
                    else:
                        training_response =
self.advanced_ai_instance.training_pipeline.train_from_text(texts,
save_checkpoints_every=CONFIG['TRAINING']['SAVE_CHECKPOINTS_EVERY'])
                        if training_response.get("status") == "success":
                            response = {"status": "success", "message": "Training initiated successfully."}
                        else:
```

```
                response = {"status": "error", "message": training_response.get("message",
"Unknown error.")}


        elif action == "provide_feedback":
            ai_response = data.get("ai_response", "")
            feedback = data.get("feedback", 0)
            if ai_response and feedback in [1, 0, -1]:
                self.advanced_ai_instance.rl_agent.process_feedback(ai_response, feedback)
                response = {"status": "success", "message": "Feedback processed."}
            else:
                response = {"status": "error", "message": "Invalid feedback data."}


        elif action == "download_kaggle_dataset":
            dataset = data.get("dataset", "")
            if dataset:
                fetcher = KaggleFetcher()
                fetcher.download_finished.connect(self.on_kaggle_download_finished)
                threading.Thread(target=fetcher.download_dataset, args=(dataset,),
daemon=True).start()
                response = {"status": "success", "message": f"Dataset {dataset} download
initiated."}
            else:
                response = {"status": "error", "message": "No dataset specified."}


        elif action == "generate_graphic":
            prompt = data.get("prompt", "")
            if prompt:
```

```
            self.advanced_ai_instance.graphic_ai.generate_graphic(prompt)
            response = {"status": "success", "message": "Image generation initiated."}
        else:
            response = {"status": "error", "message": "No prompt provided for image generation."}

    elif action == "self_program_feature":
        feature_prompt = data.get("feature_prompt", "")
        file_path = data.get("file_path", "")
        if feature_prompt and file_path:
            self.advanced_ai_instance.self_programming_ai.self_program_feature(feature_prompt, file_path)
            response = {"status": "success", "message": "Feature generation initiated."}
        else:
            response = {"status": "error", "message": "Invalid feature prompt or file path."}

    elif action == "get_training_progress":
        # Inform the user that training progress is being emitted via signals
        response = {"status": "success", "message": "Training progress is being emitted in real-time via signals."}

    elif action == "clear_training_progress":
        # Currently, there's no list to clear
        response = {"status": "success", "message": "Training progress cleared."}

    else:
```

```
            response = {"status": "error", "message": "Action not recognized."}

        except json.JSONDecodeError:
            response = {"status": "error", "message": "Invalid JSON."}
        except Exception as e:
            response = {"status": "error", "message": str(e)}
            logging.error(f"Server encountered an error: {e}")
            logging.error(traceback.format_exc())

        self.send_response(200)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(response).encode('utf-8'))

    def authenticate(self, auth_header):
        try:
            auth_type, credentials = auth_header.split(' ', 1)
            if auth_type.lower() != 'basic':
                return False
            decoded = base64.b64decode(credentials).decode('utf-8')
            username, password = decoded.split(':', 1)
            return username == self.USERNAME and password == self.PASSWORD
        except Exception as e:
            logging.error(f"Authentication failed: {e}")
            logging.error(traceback.format_exc())
            return False
```

```
    def on_kaggle_download_finished(self, dataset, success, message):
        if success:
            logging.info(f"Kaggle download finished for {dataset}: Success - {message}")
        else:
            logging.error(f"Kaggle download finished for {dataset}: Failure - {message}")

    def log_message(self, format, *args):
        return  # Suppress default logging

# ============================
# Kaggle Fetcher (Placeholder)
# ============================

class KaggleFetcher(QObject):
    download_finished = pyqtSignal(str, bool, str)

    def download_dataset(self, dataset):
        """
        Downloads a Kaggle dataset.
        """
        try:
            # Placeholder: Implement Kaggle API download logic
            logging.info(f"Starting download for Kaggle dataset: {dataset}")
            time.sleep(5)  # Simulate download time
            # Assume success
```

```
            self.download_finished.emit(dataset, True, "Download completed successfully.")
            logging.info(f"Kaggle dataset {dataset} downloaded successfully.")
        except Exception as e:
            logging.error(f"Error downloading Kaggle dataset {dataset}: {e}")
            logging.error(traceback.format_exc())
            self.download_finished.emit(dataset, False, str(e))

# ============================
# Advanced AI Orchestration
# ============================

class AdvancedAI(QObject):
    def __init__(self, google_api_key, google_cx, device):
        super().__init__()
        logging.info("Initializing AdvancedAI...")
        try:
            # Initialize tokenizer
            self.tokenizer = AutoTokenizer.from_pretrained(CONFIG['MODEL']['NAME'])
            self.tokenizer.pad_token = self.tokenizer.eos_token  # Ensure pad token is set

            # Initialize Transformer-based model with KAN layers
            config = AutoConfig.from_pretrained(CONFIG['MODEL']['NAME'])
            self.model = TransformerLanguageModel(
                config=config,
                kan_layers=CONFIG['MODEL']['KAN_LAYERS'],
                kan_output_size=CONFIG['MODEL']['KAN_OUTPUT_SIZE']
```

```
        )
        self.model.to(device)
        self.model.train()
        logging.info("TransformerLanguageModel with KAN layers initialized successfully.")

        # Initialize TrainingPipeline
        self.training_pipeline = TrainingPipeline(self.model, self.tokenizer, device, CONFIG)
        self.training_pipeline.training_finished.connect(self.on_training_finished)
        self.training_pipeline.training_progress.connect(self.on_training_progress)
        self.training_pipeline.training_error.connect(self.on_training_error)

        # Initialize ReinforcementLearningAgent
        self.rl_agent = ReinforcementLearningAgent(self.model, self.tokenizer, device,
CONFIG['REINFORCEMENT_LEARNING']['GAMMA'])

        # Initialize GraphicAI
        self.graphic_ai = GraphicAI(device=device)
        self.graphic_ai.image_generated.connect(self.on_image_generated)

        # Initialize STOPOptimizer
        utility_function = lambda solution: self.domain_specific_utility(solution)
        self.stop_optimizer = STOPOptimizer(
            role=CONFIG['STOP_OPTIMIZER']['ROLE'],
            utility=utility_function,
            lm_call_limit=CONFIG['STOP_OPTIMIZER']['LM_CALL_LIMIT'],
            max_samples_per_call=CONFIG['STOP_OPTIMIZER']['MAX_SAMPLES_PER_CALL'],
```

```
    population_size=CONFIG['STOP_OPTIMIZER']['POPULATION_SIZE'],
    mutation_rate=CONFIG['STOP_OPTIMIZER']['MUTATION_RATE'],
    crossover_rate=CONFIG['STOP_OPTIMIZER']['CROSSOVER_RATE'],
    temperature_decay=CONFIG['STOP_OPTIMIZER']['TEMPERATURE_DECAY'],
    beam_width=CONFIG['STOP_OPTIMIZER']['BEAM_WIDTH']
)
self.stop_optimizer.set_language_model(self)

# Initialize KANOptimizer
self.kan_optimizer = KANOptimizer(
    model=self.model,
    tokenizer=self.tokenizer,
    device=device,
    utility_function=self.domain_specific_utility,
    lm_call_limit=CONFIG['KAN_OPTIMIZER']['LM_CALL_LIMIT'],
    population_size=CONFIG['KAN_OPTIMIZER']['POPULATION_SIZE']
)

# Initialize SelfProgrammingAI
self.self_programming_ai = SelfProgrammingAI(device=device)
self.self_programming_ai.feature_generated.connect(self.on_feature_generated)

# Initialize SearchEngine with Google Custom Search
self.search_engine = SearchEngine(api_key=google_api_key, cx=google_cx)

logging.info("AdvancedAI initialized successfully.")
```

```
    except Exception as e:
        logging.error(f"Error initializing AdvancedAI: {e}")
        logging.error(traceback.format_exc())

def domain_specific_utility(self, solution):
    """
    Defines a domain-specific utility function.
    For example, the utility could be based on code quality, length, etc.
    Here, it's based on the length of the solution as a placeholder.
    """
    return len(solution)

def generate_response(self, conversation, user_input):
    """
    Generates a response based on the conversation and user input.
    """
    try:
        prompt = self.build_prompt(conversation, user_input)
        inputs = self.tokenizer.encode(prompt, return_tensors="pt").to(device)
        with torch.no_grad():
            outputs = self.model.generate(
                input_ids=inputs,
                max_length=512,
                num_return_sequences=1,
                no_repeat_ngram_size=2,
                do_sample=True,
```

```
                top_p=0.95,
                top_k=60,
                temperature=0.7,
                eos_token_id=self.tokenizer.eos_token_id
            )
        response = self.tokenizer.decode(outputs[0], skip_special_tokens=True)
        logging.info("AI generated response successfully.")
        return response
    except Exception as e:
        logging.error(f"Error generating response: {e}")
        logging.error(traceback.format_exc())
        return "I'm sorry, I couldn't process that."


def build_prompt(self, conversation, user_input):
    """
    Builds a prompt for the language model based on the conversation history and user input.
    """
    try:
        prompt = ""
        for turn in conversation:
            if turn['role'] == 'user':
                prompt += f"User: {turn['content']}\n"
            elif turn['role'] == 'ai':
                prompt += f"AI: {turn['content']}\n"
        prompt += f"User: {user_input}\nAI:"
```

```
            return prompt
        except Exception as e:
            logging.error(f"Error building prompt: {e}")
            logging.error(traceback.format_exc())
            return user_input

    def train_from_text(self, texts, save_checkpoints_every=5):
        """
        Initiates training from a list of texts.
        """
        try:
            dataset = TextDatasetCustom(texts, self.tokenizer, max_length=512)
            dataloader = DataLoader(dataset, batch_size=CONFIG['TRAINING']['BATCH_SIZE'], shuffle=True, collate_fn=collate_fn)
            # Placeholder: Implement validation dataloader if available
            val_dataloader = dataloader  # Using training data as validation for simplicity
            threading.Thread(target=self.training_pipeline.train, args=(dataloader, val_dataloader, CONFIG, CONFIG['TRAINING']['ITERATIVE']), daemon=True).start()
            return {"status": "success"}
        except Exception as e:
            logging.error(f"Error initiating training: {e}")
            logging.error(traceback.format_exc())
            return {"status": "error", "message": str(e)}

    def process_feedback(self, ai_response, feedback):
        """
        Processes user feedback for reinforcement learning.
```

```
        """
        self.rl_agent.process_feedback(ai_response, feedback)

    def send_crawled_text_for_training(self, crawled_texts):
        """
        Receives crawled texts and initiates training.
        """
        self.train_from_text(crawled_texts)

    def on_training_progress(self, epoch, loss, perplexity, accuracy):
        # Placeholder: Implement signal emission to GUI if needed
        logging.info(f"Training Progress - Epoch: {epoch}, Loss: {loss}, Perplexity: {perplexity}, Accuracy: {accuracy}")

    def on_training_finished(self, success, message):
        # Placeholder: Implement signal emission to GUI if needed
        logging.info(f"Training Finished - Success: {success}, Message: {message}")

    def on_training_error(self, message):
        # Placeholder: Implement signal emission to GUI if needed
        logging.error(f"Training Error: {message}")

    def on_image_generated(self, image_path):
        # Placeholder: Implement signal emission to GUI if needed
        logging.info(f"Image generated at {image_path}")
```

```
    def on_feature_generated(self, success, message):
        # Placeholder: Implement signal emission to GUI if needed
        if success:
            logging.info(f"Feature Generated Successfully: {message}")
        else:
            logging.error(f"Feature generation failed: {message}")

# ============================
# Text Dataset and Collate Function
# ============================

class TextDatasetCustom(Dataset):
    def __init__(self, texts, tokenizer, max_length=512):
        self.texts = texts
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_length,
```

```
            truncation=True,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt'
        )
        return encoding['input_ids'].squeeze(), encoding['attention_mask'].squeeze()

def collate_fn(batch):
    input_ids, attention_masks = zip(*batch)
    input_ids = torch.stack(input_ids)
    attention_masks = torch.stack(attention_masks)
    return input_ids, attention_masks

# ============================
# ChatBot GUI
# ============================

class ChatBotGUI(QMainWindow):
    def __init__(self, advanced_ai_instance, config):
        super().__init__()
        self.ai_client = AIClient(host=config['AI_SERVER_HOST'],
port=config['AI_SERVER_PORT'])
        self.conversation = []
        self.advanced_ai_instance = advanced_ai_instance
        self.config = config
        self.init_ui()
```

```
        self.load_logs()
        self.polling_timer = QTimer()
        self.polling_timer.timeout.connect(self.fetch_training_progress)

        # Connect signals
self.advanced_ai_instance.graphic_ai.image_generated.connect(self.on_image_generated)
self.advanced_ai_instance.training_pipeline.training_finished.connect(self.on_training_finished)
self.advanced_ai_instance.training_pipeline.training_progress.connect(self.on_training_progress)
self.advanced_ai_instance.training_pipeline.training_error.connect(self.on_training_error)
self.advanced_ai_instance.self_programming_ai.feature_generated.connect(self.on_feature_generated)

        self.installEventFilter(self)

    def init_ui(self):
        self.setWindowTitle("Self-Learning AI ChatBot")
        self.setGeometry(100, 100, 1200, 800)

        menubar = self.menuBar()
        file_menu = menubar.addMenu("File")
```

```
save_chat_action = QAction("Save Chat", self)
save_chat_action.triggered.connect(self.save_chat)
file_menu.addAction(save_chat_action)

load_chat_action = QAction("Load Chat", self)
load_chat_action.triggered.connect(self.load_chat)
file_menu.addAction(load_chat_action)

fetch_kaggle_action = QAction("Fetch Kaggle Dataset", self)
fetch_kaggle_action.triggered.connect(self.fetch_kaggle_dataset)
file_menu.addAction(fetch_kaggle_action)

save_logs_action = QAction("Save Logs", self)
save_logs_action.triggered.connect(self.save_logs)
file_menu.addAction(save_logs_action)

load_logs_action = QAction("Load Logs", self)
load_logs_action.triggered.connect(self.load_logs)
file_menu.addAction(load_logs_action)

exit_action = QAction("Exit", self)
exit_action.triggered.connect(self.close_application)
file_menu.addAction(exit_action)

central_widget = QWidget()
self.setCentralWidget(central_widget)
```

```
layout = QVBoxLayout()

self.chat_browser = QTextBrowser()
self.chat_browser.setStyleSheet("background-color: #FFFFFF; color: #000000;")
layout.addWidget(self.chat_browser)

input_layout = QHBoxLayout()
self.user_input = QLineEdit()
self.user_input.setPlaceholderText("Type your message here...")
self.send_button = QPushButton("Send")
self.send_button.clicked.connect(self.handle_user_input)

self.crawl_button = QPushButton("Crawl")
self.crawl_button.clicked.connect(self.web_crawl)

self.train_button = QPushButton("Train")
self.train_button.clicked.connect(self.train_model)

self.upload_file_button = QPushButton("Upload File")
self.upload_file_button.clicked.connect(self.upload_file)

self.generate_image_button = QPushButton("Generate Image")
self.generate_image_button.clicked.connect(self.generate_image)

self.self_program_button = QPushButton("Self-Program Feature")
```

```
self.self_program_button.clicked.connect(self.self_program_feature)

input_layout.addWidget(self.user_input)
input_layout.addWidget(self.send_button)
input_layout.addWidget(self.crawl_button)
input_layout.addWidget(self.train_button)
input_layout.addWidget(self.upload_file_button)
input_layout.addWidget(self.generate_image_button)
input_layout.addWidget(self.self_program_button)

layout.addLayout(input_layout)

feedback_layout = QHBoxLayout()
self.feedback_positive = QPushButton(" 👍 ")
self.feedback_positive.setToolTip("Positive Feedback")
self.feedback_positive.clicked.connect(lambda: self.provide_feedback(1))
self.feedback_neutral = QPushButton(" 😐 ")
self.feedback_neutral.setToolTip("Neutral Feedback")
self.feedback_neutral.clicked.connect(lambda: self.provide_feedback(0))
self.feedback_negative = QPushButton(" 👎 ")
self.feedback_negative.setToolTip("Negative Feedback")
self.feedback_negative.clicked.connect(lambda: self.provide_feedback(-1))
feedback_layout.addWidget(QLabel("Provide Feedback:"))
feedback_layout.addWidget(self.feedback_positive)
feedback_layout.addWidget(self.feedback_neutral)
feedback_layout.addWidget(self.feedback_negative)
```

```
        layout.addLayout(feedback_layout)

        self.training_progress = QProgressBar()
        self.training_progress.setValue(0)
        self.training_progress.setVisible(False)
        layout.addWidget(self.training_progress)

        self.metrics_display = QLabel("Training Metrics: Epoch=0, Loss=0.0, Perplexity=0.0, Accuracy=0.0")
        self.metrics_display.setStyleSheet("background-color: #E0E0E0; padding: 5px;")
        layout.addWidget(self.metrics_display)

        self.logs_browser = QTextBrowser()
        self.logs_browser.setStyleSheet("background-color: #F0F0F0; color: #000000;")
        layout.addWidget(QLabel("Application Logs:"))
        layout.addWidget(self.logs_browser)

        central_widget.setLayout(layout)

    def handle_user_input(self):
        user_input = self.user_input.text().strip()
        if user_input:
            self.append_chat(f"You: {user_input}")
            self.conversation.append({"role": "user", "content": user_input})
            self.user_input.clear()
```

```
        # Disable send button to prevent multiple submissions
        self.send_button.setEnabled(False)

        # Apply STOPOptimizer universally
        optimized_input = self.advanced_ai_instance.stop_optimizer.optimize(user_input)
        if optimized_input:
            self.append_chat(f"<i>Optimized Input: {optimized_input}</i>")
        else:
            optimized_input = user_input
            self.append_chat("<i>Unable to optimize your input. Proceeding without optimization.</i>")

        # Determine action based on optimized input
        if optimized_input.lower().startswith("generate image"):
            prompt = optimized_input[len("generate image"):].strip()
            if prompt:
                self.append_chat("<i>Generating image...</i>")
                self.advanced_ai_instance.graphic_ai.generate_graphic(prompt)
            else:
                self.append_chat("<i>Please provide a description for the image.</i>")
        elif optimized_input.lower().startswith("generate code"):
            prompt = optimized_input[len("generate code"):].strip()
            if prompt:
                self.append_chat("<i>Generating code...</i>")
                threading.Thread(target=self.ai_generate_code, args=(prompt,), daemon=True).start()
```

```
            else:
                self.append_chat("<i>Please provide a description for the code.</i>")
        else:
            request = {
                "action": "generate_response",
                "data": {"conversation": self.conversation, "user_input": optimized_input},
            }
            threading.Thread(target=self.send_request_to_server, args=(request,), daemon=True).start()

    def send_request_to_server(self, request):
        response = self.ai_client.send_request(request)
        if response.get("status") == "success":
            if "response" in response:
                ai_response = response["response"]
                self.conversation.append({"role": "ai", "content": ai_response})
                self.append_chat(f"AI: {ai_response}")
        else:
            error_msg = response.get("message", "Unknown error.")
            self.append_chat(f"Error: {error_msg}")
        # Re-enable send button after response
        self.send_button.setEnabled(True)

    def ai_generate_code(self, prompt):
        try:
            # Optimize the code generation prompt using STOPOptimizer
```

```
        optimized_prompt = self.advanced_ai_instance.stop_optimizer.optimize(prompt)
        if optimized_prompt:
            prompt_to_use = optimized_prompt
            logging.info(f"Optimized prompt for code generation: {prompt_to_use}")
        else:
            prompt_to_use = prompt
            logging.warning("Failed to optimize code generation prompt. Using original prompt.")

        request = {
            "action": "generate_response",
            "data": {"conversation": self.conversation, "user_input": prompt_to_use}
        }
        response = self.ai_client.send_request(request)
        if response.get("status") == "success" and "response" in response:
            code = response["response"]
            self.conversation.append({"role": "ai", "content": code})
            language = self.extract_language_from_prompt(prompt)
            highlighted_code = highlight_code(code, language)
            self.append_chat(f"AI: {highlighted_code}")
        else:
            error_msg = response.get("message", "Unknown error.")
            self.append_chat(f"<i>Code Generation Error: {error_msg}</i>")
    except Exception as e:
        self.append_chat(f"<i>Code Generation Error: {e}</i>")
        logging.error(f"Code Generation Error: {e}")
```

```
            logging.error(traceback.format_exc())
        finally:
            # Re-enable send button after response
            self.send_button.setEnabled(True)

    def extract_language_from_prompt(self, prompt):
        languages = DataExtractor.SUPPORTED_PROGRAMMING_LANGUAGES
        for lang in languages:
            if lang in prompt.lower():
                return lang
        return 'python'  # Default to Python if no language is detected

    def web_crawl(self):
        seed_urls_text, ok = QInputDialog.getText(self, "Enter Seed URLs", "Provide seed URLs separated by commas:")
        if ok and seed_urls_text.strip():
            seed_urls = [url.strip() for url in seed_urls_text.split(",") if url.strip()]
            if seed_urls:
                self.append_chat("<i>Started web crawling...</i>")
                self.crawler_worker = WebCrawlerWorker(
                    seed_urls=seed_urls,
                    save_dir=self.config['CRAWLED_DATA_DIR'],
                    max_depth=3,
                    max_pages=self.config['MAX_PAGES'],
                    num_threads=CONFIG['CRAWLER_THREADS'],
                    training_callback=self.advanced_ai_instance.send_crawled_text_for_training
```

```
            )
            self.crawler_worker.crawl_finished.connect(lambda: self.append_chat("<i>Web crawling completed.</i>"))
            self.crawler_worker.crawl_progress.connect(self.update_training_progress_bar)
            self.crawler_worker.start()

    def update_training_progress_bar(self, value):
        self.training_progress.setVisible(True)
        self.training_progress.setValue(value)

    def train_model(self):
        folder_path = QFileDialog.getExistingDirectory(self, "Select Training Data Folder", self.config['DATA_DIR'])
        if folder_path:
            self.training_progress.setVisible(True)
            self.training_progress.setValue(0)
            self.train_button.setEnabled(False)

            texts = []
            total_files = 0
            for root, dirs, files in os.walk(folder_path):
                for file in files:
                    file_path = os.path.join(root, file)
                    text = DataExtractor.extract_text(file_path)
                    if text:
                        # Apply data augmentation for code files
```

```
                if any(file.lower().endswith(ext) for ext in
DataExtractor.SUPPORTED_FILE_EXTENSIONS if ext.startswith('.')):
                    augmented_texts = self.advanced_ai_instance.kan_optimizer.optimize(text)
                    optimized_text =
self.advanced_ai_instance.stop_optimizer.optimize(augmented_texts)
                    if optimized_text:
                        texts.append(optimized_text)
                else:
                    optimized_text = self.advanced_ai_instance.stop_optimizer.optimize(text)
                    if optimized_text:
                        texts.append(optimized_text)
            total_files += 1

        if total_files == 0 or not texts:
            self.append_chat("<i>No valid text data found in the selected folder.</i>")
            self.training_progress.setVisible(False)
            self.train_button.setEnabled(True)
            return

        self.append_chat(f"<i>Starting training with {len(texts)} optimized text samples from
{folder_path}</i>")
        training_response = self.advanced_ai_instance.train_from_text(texts,
save_checkpoints_every=CONFIG['TRAINING']['SAVE_CHECKPOINTS_EVERY'])
        if training_response.get("status") == "success":
            self.append_chat("<i>Training initiated.</i>")
            logging.info("Model training initiated successfully.")
            self.polling_timer.start(1000)
```

```
        else:
            error_msg = training_response.get("message", "Unknown error.")
            self.append_chat(f"<i>Training Error: {error_msg}</i>")
            logging.error(f"Training Error: {error_msg}")
            self.training_progress.setVisible(False)
            self.train_button.setEnabled(True)
    else:
        self.append_chat("<i>No folder selected for training.</i>")

def upload_file(self):
    file_path, _ = QFileDialog.getOpenFileName(self, "Upload File", self.config['DATA_DIR'], "All Files (*)")
    if file_path:
        self.append_chat(f"<i>Uploading and processing file: {file_path}</i>")
        threading.Thread(target=self.extract_text_from_file, args=(file_path,), daemon=True).start()

def extract_text_from_file(self, file_path):
    try:
        text = DataExtractor.extract_text(file_path)
        if text:
            # Integrate STOPOptimizer with uploaded file training
            optimized_text = self.advanced_ai_instance.stop_optimizer.optimize(text)
            if optimized_text:
                self.append_chat(f"<i>Extracted and optimized Text from File:</i>\n{optimized_text}")
                training_response = self.ai_client.send_request({
```

```
                "action": "train",
                "data": {"texts": [optimized_text]}
            })
            if training_response.get("status") == "success":
                self.append_chat("<i>Training initiated with uploaded file.</i>")
                logging.info(f"Training initiated with uploaded file: {file_path}")
            else:
                error_msg = training_response.get("message", "Unknown error.")
                self.append_chat(f"<i>Training Error: {error_msg}</i>")
                logging.error(f"Training Error for file {file_path}: {error_msg}")
        else:
            self.append_chat("<i>Failed to optimize the extracted text from the file.</i>")
            logging.error(f"STOPOptimizer failed to optimize text from file: {file_path}")
    else:
        self.append_chat("<i>No meaningful text extracted from the file.</i>")
        logging.warning(f"No meaningful text extracted from file: {file_path}")
except Exception as e:
    self.append_chat(f"<i>Error processing file: {e}</i>")
    logging.error(f"Error processing file {file_path}: {e}")
    logging.error(traceback.format_exc())

def generate_image(self):
    prompt, ok = QInputDialog.getText(self, "Generate Image", "Enter image description:")
    if ok and prompt.strip():
        # Integrate STOPOptimizer with image generation prompt
        optimized_prompt = self.advanced_ai_instance.stop_optimizer.optimize(prompt)
```

```
        if optimized_prompt:
            prompt_to_use = optimized_prompt
            logging.info(f"Optimized prompt for image generation: {prompt_to_use}")
        else:
            prompt_to_use = prompt
            logging.warning("Failed to optimize image generation prompt. Using original prompt.")

        self.append_chat(f"<i>Generating image for: {prompt_to_use}</i>")
        self.advanced_ai_instance.graphic_ai.generate_graphic(prompt_to_use)

    def on_image_generated(self, image_path):
        if image_path and os.path.exists(image_path):
            # Ensure the image path is in a format that QTextBrowser can display
            # For local files, we need to use file:// URI scheme
            uri = f"file:///{os.path.abspath(image_path)}"
            self.chat_browser.append(f"<img src='{uri}' width='300'>")
            logging.info(f"Image generated and displayed: {image_path}")
        else:
            self.append_chat("<i>Image generation failed.</i>")
            logging.error("Image generation failed: No valid image path returned.")

    def self_program_feature(self):
        feature_prompt, ok = QInputDialog.getText(self, "Self-Program Feature", "Describe the new feature to generate:")
        if ok and feature_prompt.strip():
            file_path, _ = QFileDialog.getSaveFileName(
```

```
            self, "Select File to Integrate Feature", "auto_generated_module.py", "Python Files (*.py);;All Files (*)"
        )
        if file_path:
            # Integrate STOPOptimizer with feature generation prompt
            optimized_feature_prompt = self.advanced_ai_instance.stop_optimizer.optimize(feature_prompt)
            if optimized_feature_prompt:
                prompt_to_use = optimized_feature_prompt
                logging.info(f"Optimized feature prompt: {prompt_to_use}")
            else:
                prompt_to_use = feature_prompt
                logging.warning("Failed to optimize feature prompt. Using original prompt.")

            self.append_chat("<i>Self-Programming AI: Generating feature based on prompt...</i>")

self.advanced_ai_instance.self_programming_ai.self_program_feature(prompt_to_use, file_path)
        else:
            self.append_chat("<i>No file selected for integrating the feature.</i>")

    def on_feature_generated(self, success, message):
        if success:
            self.append_chat(f"<i>{message}</i>")
            logging.info(f"Feature generated successfully: {message}")
        else:
```

```
        self.append_chat(f"<i>Feature Generation Failed: {message}</i>")
        logging.error(f"Feature generation failed: {message}")

    def provide_feedback(self, feedback):
        """
        Sends user feedback to the AI for reinforcement learning.
        """
        last_ai_response = self.conversation[-1]["content"] if self.conversation else ""
        if last_ai_response:
            self.advanced_ai_instance.process_feedback(last_ai_response, feedback)
            feedback_text = "Positive" if feedback == 1 else ("Neutral" if feedback == 0 else "Negative")
            self.append_chat(f"<i>Feedback received: {feedback_text}</i>")
        else:
            self.append_chat("<i>No AI response to provide feedback on.</i>")

    def save_chat(self):
        if not self.conversation:
            self.append_chat("<i>No conversation to save.</i>")
            return

        file_path, _ = QFileDialog.getSaveFileName(self, "Save Chat", self.config['CHATS_DIR'], "JSON Files (*.json)")
        if file_path:
            try:
                with open(file_path, "w", encoding="utf-8") as f:
                    json.dump(self.conversation, f, indent=4)
```

```
            self.append_chat(f"<i>Chat saved to {file_path}.</i>")
            logging.info(f"Chat saved to {file_path}.")
        except Exception as e:
            self.append_chat(f"<i>Failed to save chat: {e}</i>")
            logging.error(f"Failed to save chat to {file_path}: {e}")
            logging.error(traceback.format_exc())

    def load_chat(self):
        file_path, _ = QFileDialog.getOpenFileName(self, "Load Chat", self.config['CHATS_DIR'], "JSON Files (*.json)")
        if file_path:
            try:
                with open(file_path, "r", encoding="utf-8") as f:
                    loaded_convo = json.load(f)
                self.conversation = loaded_convo
                self.chat_browser.clear()
                for msg in self.conversation:
                    sender = msg.get('role', 'user')
                    content = msg.get('content', '')
                    if sender == 'user':
                        self.append_chat(f"You: {content}")
                    elif sender == 'ai':
                        # Highlight code if present
                        if content.startswith("<pre>") and content.endswith("</pre>"):
                            self.chat_browser.append(content)
                        else:
```

```
                self.append_chat(f"AI: {content}")
            self.append_chat("<i>Chat loaded.</i>")
            logging.info(f"Chat loaded from {file_path}.")
        except Exception as e:
            self.append_chat(f"<i>Failed to load chat: {e}</i>")
            logging.error(f"Failed to load chat from {file_path}: {e}")
            logging.error(traceback.format_exc())

    def fetch_kaggle_dataset(self):
        dataset, ok = QInputDialog.getText(self, "Fetch Kaggle Dataset", "Enter Kaggle dataset identifier (e.g., username/dataset):")
        if ok and dataset.strip():
            self.append_chat(f"<i>Fetching Kaggle dataset: {dataset}</i>")
            self.kaggle_fetcher = KaggleFetcher()
            self.kaggle_fetcher.download_finished.connect(self.on_kaggle_download_finished)
            threading.Thread(target=self.kaggle_fetcher.download_dataset, args=(dataset,), daemon=True).start()

    def on_kaggle_download_finished(self, dataset, success, message):
        if success:
            self.append_chat(f"<i>Kaggle dataset {dataset} downloaded successfully.</i>")
            logging.info(f"Kaggle dataset {dataset} downloaded.")
        else:
            self.append_chat(f"<i>Error downloading Kaggle dataset {dataset}: {message}</i>")
            logging.error(f"Error downloading Kaggle dataset {dataset}: {message}")

    def save_logs(self):
```

```
    try:
        log_file = self.config['LOGS_PATH']
        if os.path.exists(log_file):
            with open(log_file, "r", encoding="utf-8", errors='ignore') as f:
                logs = f.read()
            timestamp = int(time.time())
            log_save_path = os.path.join(self.config['LOGS_DIR'], f"log_{timestamp}.txt")
            with open(log_save_path, "w", encoding="utf-8") as f:
                f.write(logs)
            self.append_chat(f"<i>Logs saved to {log_save_path}.</i>")
            logging.info(f"Logs saved to {log_save_path}.")
        else:
            self.append_chat("<i>No logs available to save.</i>")
            logging.warning("No logs available to save.")
    except Exception as e:
        self.append_chat(f"<i>Failed to save logs: {e}</i>")
        logging.error(f"Failed to save logs: {e}")
        logging.error(traceback.format_exc())


def load_logs(self):
    N = 1000
    log_file = self.config['LOGS_PATH']
    if os.path.exists(log_file):
        try:
            with open(log_file, "r", encoding="utf-8", errors='ignore') as f:
                lines = f.readlines()
```

```
                logs = ''.join(lines[-N:])
            self.logs_browser.setText(logs)
            logging.info("Application logs loaded into GUI.")
        except Exception as e:
            self.logs_browser.setText("Failed to load logs.")
            logging.error(f"Failed to load logs into GUI: {e}")
            logging.error(traceback.format_exc())
    else:
        self.logs_browser.setText("No logs available.")
        logging.warning("No logs available to load into GUI.")

def close_application(self):
    self.close()

def fetch_training_progress(self):
    response = self.ai_client.send_request({
        "action": "get_training_progress",
        "data": {}
    })
    if response.get("status") == "success":
        # Since 'training_progress_list' does not exist, inform the user accordingly
        self.append_chat(f"<i>{response.get('message')}</i>")
    else:
        logging.error(f"Failed to fetch training progress: {response.get('message', 'Unknown
error')}")
```

```
    def append_chat(self, message):
        self.chat_browser.append(message)

    def on_training_progress(self, epoch, loss, perplexity, accuracy):
        self.training_progress.setVisible(True)
        self.training_progress.setValue(int((epoch / self.config['TRAINING']['EPOCHS']) * 100))
# Assuming fixed epochs
        self.metrics_display.setText(f"Training Metrics: Epoch={epoch}, Loss={loss:.4f},
Perplexity={perplexity:.4f}, Accuracy={accuracy:.4f}")

    def on_training_error(self, message):
        self.train_button.setEnabled(True)
        self.training_progress.setVisible(False)
        self.append_chat(f"<i>Training Error: {message}</i>")
        logging.error(f"Training Error: {message}")

    def on_training_finished(self, success, message):
        self.train_button.setEnabled(True)
        self.training_progress.setVisible(False)
        if success:
            self.append_chat(f"<i>{message}</i>")
            logging.info(f"Training finished: {message}")
        else:
            self.append_chat(f"<i>Training Error: {message}</i>")
            logging.error(f"Training Error: {message}")

    def eventFilter(self, source, event):
```

```
        if event.type() == QEvent.Close:
            self.auto_save_on_close()
        return super().eventFilter(source, event)

    def auto_save_on_close(self):
        try:
            logging.info("Auto-saving model to both checkpoints before closing.")

self.advanced_ai_instance.training_pipeline.save_model(self.advanced_ai_instance.trainin
g_pipeline.checkpoint_1)

self.advanced_ai_instance.training_pipeline.save_model(self.advanced_ai_instance.trainin
g_pipeline.checkpoint_2)
            logging.info("Auto-save completed successfully.")
        except Exception as e:
            logging.error(f"Error during auto-save on close: {e}")
            logging.error(traceback.format_exc())

    def ai_generate_code(self, prompt):
        try:
            # Optimize the code generation prompt using STOPOptimizer
            optimized_prompt = self.advanced_ai_instance.stop_optimizer.optimize(prompt)
            if optimized_prompt:
                prompt_to_use = optimized_prompt
                logging.info(f"Optimized prompt for code generation: {prompt_to_use}")
            else:
                prompt_to_use = prompt
```

```
            logging.warning("Failed to optimize code generation prompt. Using original prompt.")


        request = {
            "action": "generate_response",
            "data": {"conversation": self.conversation, "user_input": prompt_to_use}
        }
        response = self.ai_client.send_request(request)
        if response.get("status") == "success" and "response" in response:
            code = response["response"]
            self.conversation.append({"role": "ai", "content": code})
            language = self.extract_language_from_prompt(prompt)
            highlighted_code = highlight_code(code, language)
            self.append_chat(f"AI: {highlighted_code}")
        else:
            error_msg = response.get("message", "Unknown error.")
            self.append_chat(f"<i>Code Generation Error: {error_msg}</i>")
    except Exception as e:
        self.append_chat(f"<i>Code Generation Error: {e}</i>")
        logging.error(f"Code Generation Error: {e}")
        logging.error(traceback.format_exc())
    finally:
        # Re-enable send button after response
        self.send_button.setEnabled(True)

# ============================
```

```
# Web Crawler Worker (Placeholder)
# ============================

class WebCrawlerWorker(QObject):
    crawl_finished = pyqtSignal()
    crawl_progress = pyqtSignal(int)

    def __init__(self, seed_urls, save_dir, max_depth, max_pages, num_threads,
training_callback):
        super().__init__()
        self.seed_urls = seed_urls
        self.save_dir = save_dir
        self.max_depth = max_depth
        self.max_pages = max_pages
        self.num_threads = num_threads
        self.training_callback = training_callback
        self.pages_crawled = 0
        self.lock = threading.Lock()

    def start_crawling(self):
        # Placeholder: Implement multi-threaded web crawling logic
        try:
            logging.info("Starting web crawling...")
            for url in self.seed_urls:
                if self.pages_crawled >= self.max_pages:
                    break
```

```
            # Simulate crawling
            time.sleep(1)
            self.pages_crawled += 1
            self.crawl_progress.emit(int((self.pages_crawled / self.max_pages) * 100))
        # After crawling, invoke training callback
        crawled_texts = ["Sample crawled text data."]  # Placeholder
        self.training_callback(crawled_texts)
        self.crawl_finished.emit()
        logging.info("Web crawling completed successfully.")
      except Exception as e:
        logging.error(f"Error during web crawling: {e}")
        logging.error(traceback.format_exc())

  def start(self):
    threading.Thread(target=self.start_crawling, daemon=True).start()

# ============================
# Resource Monitoring
# ============================

def monitor_resources():
  process = psutil.Process(os.getpid())
  while True:
    mem = process.memory_info().rss / (1024 ** 2)  # Memory in MB
    cpu = psutil.cpu_percent(interval=1)
    logging.info(f"Memory Usage: {mem:.2f} MB, CPU Usage: {cpu:.2f}%")
```

```
        time.sleep(60)  # Log every 60 seconds


# ============================
# Main Function
# ============================


def start_ai_server(advanced_ai_instance, config):
    """
    Starts the AI server to handle HTTP requests.
    """
    try:
        AIServer.advanced_ai_instance = advanced_ai_instance
        server = HTTPServer((config['AI_SERVER_HOST'], config['AI_SERVER_PORT']), AIServer)
        server_thread = threading.Thread(target=server.serve_forever, daemon=True)
        server_thread.start()
        logging.info("AI Server thread started.")
        print(f"AI Server started and listening on
{config['AI_SERVER_HOST']}:{config['AI_SERVER_PORT']}")
        return server
    except Exception as e:
        logging.error(f"Failed to start AI Server: {e}")
        logging.error(traceback.format_exc())
        sys.exit(1)


def main():
    # Define the device
```

```
    global device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    logging.info(f"Using device: {device}")

    # Ensure that 'data/kaggle.json' exists or handle its creation separately
    if not os.path.exists(CONFIG['KAGGLE_JSON_PATH']):
        logging.warning(f"{CONFIG['KAGGLE_JSON_PATH']} does not exist. Please provide your Kaggle API credentials.")
        # Optionally, prompt the user to provide it or create an empty file
        # For security reasons, avoid creating an empty file; better to notify the user.

    # Initialize AdvancedAI with correct CONFIG keys
    advanced_ai_instance = AdvancedAI(
        google_api_key=CONFIG['GOOGLE_API_KEY'],
        google_cx=CONFIG['GOOGLE_CX'],
        device=device
    )
    # Start AI Server
    server = start_ai_server(advanced_ai_instance, CONFIG)
    # Start resource monitoring in a separate thread
    threading.Thread(target=monitor_resources, daemon=True).start()
    # Initialize and start the GUI
    app = QApplication(sys.argv)
    window = ChatBotGUI(advanced_ai_instance, CONFIG)
    window.server = server
    window.show()
```

```
    sys.exit(app.exec_())


if __name__ == "__main__":
    # Download NLTK data if not already present
    try:
        nltk.data.find('tokenizers/punkt')
    except LookupError:
        nltk.download('punkt')
    try:
        nltk.data.find('corpora/stopwords')
    except LookupError:
        nltk.download('stopwords')


    main()
```

Explanation of Key Fixes and Enhancements

1. Correct Initialization of KANLayer Instances

Problem: All KANLayer instances were initialized with input_size=config.hidden_size (e.g., 1024) and output_size=100. While the first layer correctly transforms the hidden state from 1024 to 100, subsequent layers expected 1024 as input but received 100, causing a shape mismatch during matrix multiplication.

Solution:

- **First KANLayer:** Initialize with input_size=config.hidden_size and output_size=kan_output_size.
- **Subsequent KANLayers:** Initialize with input_size=kan_output_size and output_size=kan_output_size.

Implementation:

python

Copy code

```
self.kan_layers = nn.ModuleList(
  [KANLayer(config.hidden_size, kan_output_size)] +
  [KANLayer(kan_output_size, kan_output_size) for _ in range(kan_layers - 1)]
)
```

This ensures that each KANLayer receives the correct input size, preventing tensor shape mismatches.

2. Enhanced Logging in KANLayer and TransformerLanguageModel

Problem: Insufficient logging made it challenging to trace tensor shapes and identify where mismatches occurred.

Solution:

- **Add Debug Logs:** Insert logging statements to output tensor shapes at critical points within the forward methods.

Implementation: In KANLayer:

python

Copy code

```
def forward(self, x):
  # x: (batch_size * seq_length, input_size)
  matmul_result = torch.matmul(x, self.a.t())  # (batch_size * seq_length, num_functions)
  logging.debug(f"matmul_result shape: {matmul_result.shape}")

  u = torch.sin(matmul_result + self.b)  # (batch_size * seq_length, num_functions)
  logging.debug(f"u shape after sin: {u.shape}")

  v = torch.matmul(u, self.c)  # (batch_size * seq_length, output_size)
  logging.debug(f"v shape: {v.shape}")
```

```
    return v  # (batch_size * seq_length, output_size)
```

In TransformerLanguageModel:

python

Copy code

```
def forward(self, input_ids, attention_mask=None, labels=None):
    transformer_outputs = self.transformer(input_ids=input_ids,
                                           attention_mask=attention_mask,
                                           labels=labels,
                                           output_hidden_states=True)
    hidden_states = transformer_outputs.hidden_states[-1]  # Last layer hidden states
    logging.debug(f"Hidden states shape: {hidden_states.shape}")

    # Pass through KAN layers
    kan_output = hidden_states  # (batch_size, seq_length, hidden_size)
    for idx, kan_layer in enumerate(self.kan_layers, 1):
        batch_size, seq_length, hidden_size = kan_output.size()
        kan_output = kan_output.view(batch_size * seq_length, hidden_size)  # (batch_size*seq_length, hidden_size)
        logging.debug(f"Before KANLayer {idx}, kan_output shape: {kan_output.shape}")
        kan_output = kan_layer(kan_output)  # (batch_size*seq_length, kan_output_size)
        logging.debug(f"After KANLayer {idx}, kan_output shape: {kan_output.shape}")
        kan_output = kan_output.view(batch_size, seq_length, -1)  # (batch_size, seq_length, kan_output_size)

    logits = self.final_linear(kan_output)  # (batch_size, seq_length, vocab_size)
    logging.debug(f"Logits shape: {logits.shape}")
```

```
    loss = transformer_outputs.loss

    return {"loss": loss, "logits": logits}
```

These debug logs will help you trace the dimensions of tensors during each operation, making it easier to identify and rectify any mismatches.

3. Ensuring Consistent Tokenizer and Model Configuration

Problem: Mismatches between the tokenizer's vocab_size and the model's vocab_size can lead to unexpected behaviors.

Solution:

- **Use Tokenizer's vocab_size:** Reference the tokenizer's vocab_size directly to initialize metrics and layers, ensuring consistency.

Implementation:

python

Copy code

```
vocab_size = self.tokenizer.vocab_size  # Use tokenizer's vocab size to avoid mismatches

self.accuracy_metric = Accuracy(task='multiclass',
num_classes=vocab_size).to(self.device)
```

4. Additional Recommendations

- **Avoid Naming Conflicts:** Ensure there's no local file named logging.py in your project directory (C:\Users\rober\source\repos\self_learning_ai\). Such a file can overshadow Python's standard logging module, leading to unexpected errors.
- **Install All Required Dependencies:**
    - Use the following command to install all necessary Python libraries:

bash

Copy code

```
pip install PyPDF2 python-docx pytesseract pillow beautifulsoup4 pandas torch torchvision torchaudio transformers diffusers pygments nltk psutil PyQt5
```

- **Set Up Tesseract OCR:**
    - **Windows:** Download the installer from Tesseract at UB Mannheim.

- **macOS:** Use Homebrew:

bash

Copy code

brew install tesseract

- **Linux (Debian/Ubuntu):**

bash

Copy code

sudo apt-get update

sudo apt-get install tesseract-ocr

- **Set Environment Variables for Sensitive Information:**
  - Avoid hardcoding sensitive information like API keys in your script. Instead, set them as environment variables:
    - **Windows:**

cmd

Copy code

set GOOGLE_API_KEY=your_actual_api_key

set GOOGLE_CX=your_custom_search_engine_id

    - **macOS/Linux:**

bash

Copy code

export GOOGLE_API_KEY=your_actual_api_key

export GOOGLE_CX=your_custom_search_engine_id

  - Modify your CONFIG to fetch these variables:

python

Copy code

"GOOGLE_API_KEY": os.getenv("GOOGLE_API_KEY", "YOUR_GOOGLE_API_KEY"),

"GOOGLE_CX": os.getenv("GOOGLE_CX", "YOUR_CUSTOM_SEARCH_ENGINE_ID"),

- **Monitor Resource Usage:**
    - The application includes resource monitoring that logs CPU and memory usage every 60 seconds. Ensure that your system has sufficient resources to handle the application's demands.
- **Validate Model Configuration:**
    - Ensure that config.hidden_size correctly reflects the transformer's hidden size (e.g., 1024 for codegen-350M-multi).
    - Avoid manually overriding hidden_size and vocab_size in the configuration unless necessary.

Conclusion

By implementing the above fixes and enhancements, your application should no longer attempt to allocate an excessive amount of memory, thereby preventing the RuntimeError related to tensor shape mismatches. Additionally, enhanced logging will facilitate easier debugging and monitoring of your model's behavior.

If you continue to encounter issues or have further questions, feel free to ask!